Code 582

Flight Software Branch

C CODING STANDARD

Flight Software Branch – Code 582

Version 1.1 – 11/29/05

582-2000-005

Goddard Space Flight Center

Greenbelt, Maryland

National Aeronautics and
Space Administration

FORWARD AND UPDATE HISTORY

This document presents a set of ANSI C language coding standards recommended by the NASA Goddard Space Flight Center Flight Software Branch.

Version	Date	Description	Affected Pages
1.0a	09/07/00	Initial release	All
1.0b	11/05/03	Implementation of: DCR #18 (Prohibit directory information in #include) DCR #25 (Comments stating variable's units of measure) Layout changes + new cover page	 18 (item (9)) 12 (item (3)) All
1.0c	10/29/04	DCR #86 (Header file function prototypes should be marked as 'extern') DCR #87 (Update ANSI C Standard Reference)	14 (Item (7), top of page) 9 (Item (1), top of page)
1.0d	06/20/05	DCR #14 no change history in file headers	3.4(6), 2.3.1 (1)
1.1	11/29/05	DCR #140 (Compiler Settings with COTS Software (use of -Werror))	3.1 (3)

CONTENTS

1 INTRODUCTION

This document presents a set of ANSI C language coding standards recommended by the NASA Goddard Space Flight Center Flight Software Branch.

There are two primary goals for these coding standards:

- Make the C code readable, so it can be easily understood by other programmers in the branch
- Make the C language easier to use, for example by controlling the global name space

This standard is project independent. If there are project-specific C coding issues, it is recommended that a project-specific amendment be written.

Coding standards are often thought of as a nuisance and a restriction by many programmers. The exact opposite is actually the case. Standards relieve the programmer from wasting time on the mundane parts of coding allowing the programmer to focus on design issues (the fun stuff). In a team environment standards provide uniformity in programming style which aide in communication. Standards provide a legend into everyone's code so anytime one needs to discuss his/her code with another team member no time is wasted on getting oriented. Code reviews can focus on what the code is doing and the content of the comments. Coding standards make the code easier to read, understand, and to maintain.

We acknowledge that writing code that is easily readable and works takes more time than simply writing code that works. However, the time saved by the maintenance phase, and by allowing reuse in other projects, more than makes up for the time spent writing. This standard explicitly favors ease of reading over ease of writing.

This standard sets the policy for the appropriate use of "C" language constructs, indentation, brace placement, commenting, variable declaration, variable naming, and white space usage. In many cases specific use of valid "C" language constructs are restricted to prevent common logical and syntactical errors.

The term "layout" refers to the use of white space to arrange code in the source file.

The standards described in this document are either mandatory or discretionary. Mandatory standards are requirements that must be followed. These are indicated by the word **shall**. Discretionary standards are guidelines that allow some judgment or personal choice by the programmer. These standards are indicated by the use of the word **should**. A programmer should have a good reason when they choose to disregard a discretionary standard. Where possible, this document gives examples of acceptable deviations. A mandatory standard can be waived by the development lead for specific cases, where appropriate.

In this document, paragraphs with a number (n) are standards; other paragraphs are explanatory text.

The primary mode of reading code is assumed to be on a computer screen, with an editor that provides syntax colorization and code navigation. There will be a few times, primarily at code reviews, when code will be printed out. The coding standards reflect this bias towards on-screen viewing.

2 FILE ORGANIZATION

This section describes the desired source file structure of the software.

2.1 UNITS

(1) Code shall be structured as *units*, or as stand-alone header files.

(2) A unit shall consist of a single header file (.h) and one or more body (.c) files. Collectively the header and body files are referred to as the source files.

(3) A unit header file shall contain all pertinent information required by a client unit. A unit's client needs to access only the header file in order to use the unit.

(4) The unit header file shall contain *#include* statements for all other headers required by the unit header. This lets clients use a unit by including a single header file.

(5) The unit body file shall contain an *#include* statement for the unit header, before all other *#include* statements. This lets the compiler verify that all required *#include* statements are in the header file.

(6) A body file shall contain only functions associated with one unit. One body file may not provide implementations for functions declared in different headers.

(7) All client units that use any part of a given unit U shall include the header file for unit U; this ensures that there is only one place where the entities in unit U are defined. Client units may call only the functions defined in the unit header; they may not call functions defined in the body but not declared in the header. Client units may not access variables declared in the body but not in the header.

A *component* contains one or more units. For example, a math library is a component that contains multiple units such as vector, matrix, and quaternion.

Stand-alone header files do not have associated bodies; for example, a common types header does not declare functions, so it needs no body.

Some reasons for having multiple body files for a unit:

- Part of the body code is hardware or operating system dependent, but the rest is common.
- The files are too large.
- The unit is a common utility package, and some projects will only use a few of the functions. Putting each function in a separate file allows the linker to exclude the ones not used from the final image.

2.1.1 Header include rationale

This standard requires a unit's header to contain *#include* statements for all other headers required by the unit header. Placing *#include* for the unit header first in the unit body allows the compiler to verify that the header contains all required *#include* statements.

An alternate design, *not* permitted by this standard, allows no *#include* statements in headers; all *#include*s are done in the body files. Unit header files then must contain *#ifdef* statements that check that the required headers are included in the proper order.

One advantage of the alternate design is that the *#include* list in the body file is exactly the dependency list needed in a makefile, and this list is checked by the compiler. With the standard design, a tool must be used to generate the dependency list. However, all of the branch recommended development environments provide such a tool.

A major disadvantage of the alternate design is that if a unit's required header list changes, each file that uses that unit must be edited to update the *#include* statement list. Also, the required header list for a compiler library unit may be different on different targets.

Another disadvantage of the alternate design is that compiler library header files, and other third-party files, must be modified to add the required *#ifdef* statements.

A different common practice is to include all system header files before any project header files, in body files. This standard does not follow this practice, because some project header files may depend on system header files, either because they use the definitions in the system header, or because they want to override a system definition. Such project header files should contain *#include* statements for the system headers; if the body includes them first, the compiler does not check this.

2.2 FILE NAMING CONVENTIONS

(1) The filename shall be the unit name or an abbreviation of the unit name. If an abbreviation is used, it shall be the same abbreviation used in global identifier prefixes (see Section 3.2), and shall be documented in an abbreviation list.

For example, a math unit might use the prefix "math_"; the file names would be "math.h" and "math.c".

(2) Header files shall have an extension of ".h" and body files shall have an extension of ".c."

(3) A unit's header file and body file shall have the same name except for the extension. If a unit has multiple body files, the body file names shall start with the unit name or abbreviation.

(4) Filenames may be mixed case, but shall be case-insensitive unique within the branch library. This avoids confusion, and is necessary on case-insensitive file systems such as Microsoft Windows. When files are moved between systems, the filename case must be preserved, since *#include* statements rely on it.

2.3 HEADER FILES

(1) Header files shall not generate object code (data or code).

(2) Header files shall contain a set of preprocessor directives to ensure that they are not included more than once:

```
/* header prologue */

#ifndef _filename_
#define _filename_
.
header file contents
.
#endif /* _filename_ */
```

where *filename* is the header file name without the ".h" extension. This macro is mixed case (to match the file name), which is an exception to the macro naming convention.

Note that compiler library or third-party headers may not conform to this convention. Such files will either have to be modified or treated specially in body files.

(3) A header file shall contain the following sections in the order shown in Figure 1 and described below. If a section is not required then the block comment banner starting it does not need to be present. This ensures that header files are consistent, to enable other programmers to easily locate function and data declarations.

1 The file shall be introduced by a block comment that contains a header file prologue (see Section 2.3.1).

2 Header file inclusions shall be placed in the *Include* section, immediately after the multiple-file inclusion guard macro. ANSI C system library include files shall be specified first, followed by user-defined includes.

3 *#define* statements shall be placed within the *Macro Definitions* section.

4 *typedef* statements shall be placed within the *Type Definitions* section.

5 Exported data shall be placed within the *Exported Data* section.

6 The function prototypes shall be placed in the *Exported Functions* section. Function prototypes shall be documented by a block comment.

FIGURE 1. Example Header File "vec3d.h"

```
/*
** File:
** $Id:$
**
** Purpose:
** Unit specification for 3 dimensional vectors of type double.
**
** References:
** [1] math.doc: Math Documentation
**
** Notes:
** Temporary variables are used when needed to allow the Result
** to be the same variable as one of the operands.
```

```
*/
#ifndef _vec3d_
#define _vec3d_

/*
** Includes
*/

/*
** Macro Definitions
*/

#define X 0
#define Y 1
#define Z 2

/*
** Type Definitions
*/

typedef struct
{
   double Comp[3];
} Vec3d; /* See [1] Data Representation Section */

/*
** Exported Functions
*/

/*
** Add two Vec3d vectors.
** Result = Left + Right.
*/
void Vec3d_Add (Vec3d *Result, const Vec3d *Left, const Vec3d *Right);

/*
** Divide each element of a Vec3d vector by a double scalar.
** Result = Left / Scalar.
**
** Notes:
** Scalar must be a non-zero floating point number of type double,
** otherwise, divide by zero error will occur.
*/
void Vec3d_DivByScalar (Vec3d *Result, const Vec3d *Left, double
Scalar);

/*
** Compute the inner product (dot product) of two Vec3d vectors.
** Returned double = Left .dot. Right.
**
*/
double Vec3d_Dot(const Vec3d *Left, const Vec3d *Right);

#endif /* _vec3d_ */
```

2.3.1 Header File Prologue

(1) The header file prologue is a block comment in column 1 at the very start of the file, and should contain the following items. However, if any item is empty, it may be omitted. See Figure 1 for an example.

1 **Filename**: give the filename, not including the directory path. This information should be automated by the configuration management system. For example the CVS directive `$ID$` will provide the filename.

2 **Purpose**: A short statement that describes the purpose of the code in the file.

3 **Design Notes**: List design decisions that apply to all functions in the file. If possible, give a brief justification for each decision; for example, list the alternatives and say why they were rejected. Longer justifications may be given in the component document; the source code cannot contain complete documentation for complex units.

4 **References**: List all applicable documentation references. Identify each reference to a level of detail that allows the information to be found, but not to a level that will likely change. Provide version numbers when applicable. The component document should be the first reference.

5. **Revision Date**: The date of the last file revision, unless using a CM tool that maintains this information.

6. **Revision Label**: The last revision identifier, unless using a CM tool that maintains this information.

7. **Revision History**: For each previous revision, a revision identifier and description of the revision, unless using a CM tool that maintains this information.

Items 5 thru 7 should be kept in the configuration management system, rather than in the source file. Keeping this information in the source file will cause spurious conflicts when merging different branches. However, if the CM system being used does not support these functions, the information should be kept in the source file.

The “component document” is described in TBD.

2.4 BODY FILES

A body file contains the implementation of the data and function specifications defined in a header file.

(1) It is suggested that a body file should not exceed 1000 lines (20 pages) of code and comments.

(2) Functions that are not described in the header file shall have a block comment description.

(3) Each function shall be terminated by a comment including the function name that indicates the end of the function.

(4) A body file shall contain the following sections (see Figure 2 for an example body file). If a section is empty then the block comment banner does not need to be present.

1 A body file shall be introduced by a block comment that contains the file prologue, using the same format as the header file prologue (Section 2.3.1). References and design decisions documented in the header shall not be repeated.

2 Any header file *#include* statements shall be specified next. The unit header file shall be listed first, followed by ANSI standard header files, followed by system library headers, followed by other headers.

3 *#define* statements shall be placed in the *Macro Definitions* section.

4 Type definitions that have file scope shall be placed within the *Type Definitions* section.

5 Global data objects that are exported (created in this file and used by other files) shall be placed in the *Exported Global Data* section.

6 File scope data objects shall be placed in the *File Data* section.

7 Prototypes for file scope functions shall be placed in the *Local Function Prototypes* section.

8 Function definitions shall be placed in the *Function Definitions* section.

9 The last item in each file shall be an end of file comment. This comment assures a reader that a file listing is complete.

FIGURE 2. Example Body File “vec3d.c”

```
/*
** File:
** $Id:$
**
** Purpose:
** Implementation of the Vec3d unit.
**
*/

/*
** Include section
*/

#include "vec3d.h"

/*
** Macro Definitions
*/

/*
** Type Definitions
```

```
*/

/*
** Exported Global Data
*/

/*
** File Data
*/

/*
** Local Function Prototypes
*/

/*
** Function definitions
*/

void Vec3d_Add (Vec3d *Result, const Vec3d *Left, const Vec3d *Right)
{
   Result->Comp[X] = Left->Comp[X] + Right->Comp[X];
   Result->Comp[Y] = Left->Comp[Y] + Right->Comp[Y];
   Result->Comp[Z] = Left->Comp[Z] + Right->Comp[Z];
} /* End Vec3d_Add() */

void Vec3d_DivByScalar (Vec3d *Result, const Vec3d *Left, double
Scalar)
{
   Result->Comp[X] = Left->Comp[X] / Scalar;
   Result->Comp[Y] = Left->Comp[Y] / Scalar;
   Result->Comp[Z] = Left->Comp[Z] / Scalar;
} /* End Vec3d_DivByScalar() */

double Vec3d_Dot(const Vec3d *Left, const Vec3d *Right)
{
   return Left->Comp[X] * Right->Comp[X] +
      Left->Comp[Y] * Right->Comp[Y] +
      Left->Comp[Z] * Right->Comp[Z];
} /* End Vec3d_Dot() */

/* end of file */
```

3 GENERAL CODING STANDARDS

3.1 LANGUAGE

(1) The programmer shall only use ISO/IEC 9899:1990 – "ANSI C" statements unless a waiver is obtained, and the non-ANSI statements are fully documented and are absolutely necessary.

Note that this means C++ style comments are not permitted.

This standard does not define who grants the waiver; see the software process standard.

(2) Compilers shall be configured to enforce the ANSI standard. Table 1 gives the compiler switches required for several compilers.

Table 1 - Required Compiler switch settings

Compiler	Switches
GNU C	`–Wall –Wstrict-prototypes –pedantic –ansi –Werror`

(3) All code should compile without warnings. However, it is sometimes impossible to eliminate warnings; such cases should be documented with comments in the code.

It is acceptable to relax the switches to avoid warnings in 3rd party code. When this is done the switch changes must be localized to as few places as possible.

(4) A standard set of options shall be used for each compiler/target.

Some compiler options can change the meaning of code; for example, whether *char* is signed or unsigned. Other options change the layout of structures. These options must be the same for all files in a single image.

Some options can be file-specific, such as the optimization level.

(5) All source files shall be edited using printable ASCII characters, with no tabs or form feeds.

(6) Source lines shall not exceed 120 characters in length. We want to encourage long meaningful identifier names; this is easier to accommodate if lines are longer than 80 characters.

3.2 IDENTIFIER NAMING CONVENTIONS

The proper naming of types, functions and variables can increase the readability and understanding of the software. Poorly named functions and variables can add a great deal of misunderstanding and confusion. Careful consideration should be given to how others will perceive the name, and what they will think it means. A function or variable's name should provide insight into its use and purpose.

In addition, global identifiers must be unique within the library; we cannot have two different units both declaring a function named "compute". Adding the unit name as a prefix to each global identifier solves this problem.

(1) Global identifiers shall be prefixed by a unique abbreviation of the unit name and an underscore (_). Identifiers that are local to a file shall not have the unit name prefix. This convention eliminates global name collisions, and helps the reader locate where the identifier is declared.

(2) The library shall maintain a list of approved abbreviations.

(3) Identifiers should consist of more than one character. Valid exceptions are for variables used in local indexing operations (e.g., i, j, k) and very mathematical code where the identifier matches the standard symbol used in an algorithm or equation.

(4) All names declared by *#define* statements shall be all uppercase letters. The file inclusion guard used by header files is an exception to this rule.

(5) All enumeration constants shall be uppercase.

(6) Capitalization of ANSI C reserved words shall be as defined by the ANSI standard.

(7) The capitalization of other identifiers should be mixed case, with the first letter of each component word capitalized. Underscores should be used only to separate the unit abbreviation prefix.

This style is often called `MixedCaseNoUnderscores`

(8) The names of any two identifiers shall have unique spellings and not merely be distinguished by case.

(9) An initial underscore shall not be used for any user defined names. This prefix is reserved for use by system library functions and definitions.

(10) A boolean variable, or a function that returns a boolean result, should be named so that the name asserts a condition that is true when the value is ‘true’ (non-zero): TapeRecorderOn, PowerOff. Alternatively, the name should express an interrogatory with a clear answer: IsTapeRecorderOn, IsPowerOff, where ‘yes’ is represented by ‘true’.

3.3 INDENTATION AND VISUAL STANDARDS

It is easier to combine code into a system, and to maintain the code, if all of the code conforms to a particular style. However, naming styles and indentation styles are often very personal, and can be hard to change. For this reason, this standard does not mandate a particular style; programmers are free to continue using their own style. However, the standard does give detailed style recommendations (using “should” instead of “shall”), for programmers who wish to conform to a common style.

(1) An example of the recommended indentation standard:

```
/*
** Check the components of x:
*/

for (x = 0; x < 3; x++)
{
   if (Variable[x] == 0)
   {
      Variable[x] = Variable[x+3];
   }
} /* end of for x */

...next statement;
```

Note that there is no comment on the brace closing the if statement; it is clear where the opening brace is, so a comment would just be clutter.

(2) The general coding style shall be consistent within each single file.

(3) All code shall be indented a minimum of three spaces for each indentation level.

(4) Statements at the same logical nesting level shall be at the same indentation level.

(5) The indentation of a long or short comment inside a function shall be the same as the code it describes.

(6) There should be only one statement per line.

(7) Blank lines should be used between blocks of code that are functionally distinct.

(8) Braces delineating compound statements should be on separate lines, with the opening and closing brace at the same indentation as the surrounding statements. The closing brace should be commented, unless it is obvious where the matching opening brace is.

(9) In expressions, operators should be surrounded by blanks, but not operators that compose primaries, specifically ".", "->"; and not between unary operators and their operands.

```
A = B->bar + C.foo;
ptr++ = FillByte;
foo = --I;
```

3.4 COMMENTS

Comments are important in developing readable and maintainable code. Programmers should include comments whenever it is difficult to understand the code without the comments. However, one should avoid over-commenting as this only hides the real code, and makes it harder to debug, modify, and maintain.

Comments can be classified by size:

- Long or block comments are those that can not fit on a single 80 character line;
- Short comments are those that can fit on a single 80 character line, but leave no room for code. A short comment can sometimes be difficult to locate, the programmer should consider using a block comment instead of a short comment.
- Same-line comments are small enough to include on the same line as the code that the comment supports.

(1) Same-line comments shall always be to the right of the described code. Within a function all of the same-line comments should start in the same column.

(2) Functional blocks of code should have a long comment before the actual code instead of placing a comment on each line. This comment should describe the basic purpose of the code. The programmer should use complete English sentences.

```
/*
** This is an example of long block comment. The start
** and end of the comment should be separated from the
** code by white space.
*/
```

(3) A variable's units of measurement (if any) should be stated in a comment.

(4) Comments should not simply say the same thing the code does, as this only serves to obscure required comments. Describe why something is being done first, and only describe what is being done if it isn't obvious by the C code itself. The following example shows an unnecessary comment:

```
x += 1; /* Add 1 to x */ unnecessary!
```

(5) If PDL (Program Design Language) is used during development, it shall be deleted when the final code is written. A PDL summary of the C code does not provide any new information.

(6) (deleted).

(7) Old versions of the code shall not be maintained in the comments; use a configuration management tool instead.

Some alternate coding practices try to maintain change information on a per-line basis. For each change the right most columns are used to identify whether a line has been effected by a particular change. The file prologue identifies the change number and each modified line references the file prologue change identifier. If significant changes are made to a file this practice makes it hard to read and understand the current logic. This practice is unnecessary with a configuration management tool that provides source file differences between versions.

3.5 VARIABLE SCOPE

Some variables are only visible within a function (function scope); others are visible to some or all the functions in a file (file scope); still others are visible to any function in the system (global scope). The use of global variables can result in couplings between parts of a program that may not be obvious, and their use should be minimized.

However, in a real time system it is common for large amounts of data that are computed by one component to be needed by one or more other components, and global variables are used for time and space efficiency.

(1) Exported global data for a component should be combined into a few global data structures, rather than many individual data items.

(2) Only the unit that declares a global variable should write to that variable.

(3) Variables with file scope should be gathered into a file scope data structure. There should be very few individual file scope variables.

(4) File scope variables shall be declared *static* unless the address of the variable is needed in the link map. Names of file scope variables not declared *static* shall start with the unit name prefix. Variables whose addresses are in the link map can be examined or set by address when the system is in operation.

(5) Function scope variables shall not have the same name as a file scope variable in the same file. This avoids confusion.

(6) The justification for a static function scope variable shall be stated in a comment. The comment should explain why a file scope data structure is not used instead.

3.6 FUNCTION PROTOTYPES

(1) A function prototype, for either a global function in a header file, or a local function in a body file, should have the following format:

```
/*
** description, including function purpose and how each argument is
** used.
*/
[static] type FunctionName ( type Arg_1,
                              type Arg_2,
                              type Arg_n)
```

If the arguments all fit on the same line as the function name, they should be placed there.

If the type is long or complex, it may be placed on the line before the function name.

The description comment should make it clear what the function does, and how it uses or affects each argument. For simple functions with good names, the descriptive comment can be empty (for example, *vec3d_mult_scalar* in Figure 1 has no comment). For a very complex function, the description should just reference the full component documentation.

(2) The return value of all functions shall be explicitly typed. Functions that do not return values shall be declared to be of type *void.*

(3) File scope functions shall be declared as *static*, unless the function address is needed in the link map. File scope functions not declared *static* shall have a comment justifying this, and shall have the global identifier prefix.

(4) Each argument shall be explicitly declared in the argument list, with a name and a type. If a function does not require any arguments its argument list shall be declared *void*.

(5) Variable arguments shall not be used. An exception is allowed for the standard C library functions *printf* and *scanf* when used in ground or test code; however, they shall not be used in flight code.

(6) The keyword *const* shall be used to modify the arguments wherever appropriate

(7) The preferred approach for global scope function prototypes is to omit the keyword *extern*; however, it may be included if that is the developer's preference.

3.7 FUNCTION DEFINITIONS

(1) A function definition in a body file shall have the following format:

```
[static] type FunctionName ( type Arg_1,
                              type Arg_2,
                              type Arg_n)
/* optional prototype descriptive comment */
{
   /* implementation overview */

   variable declarations

   statements

}  /* end FunctionName */
```

(2) The prototype definition comment is required if there is no corresponding prototype in a header file or body file; it shall be empty if there is a corresponding prototype.

(3) The function prototype in the definition shall have the same layout as the corresponding prototype.

(4) The argument names in the function definition shall be the same as in the corresponding prototype.

(5) The opening brace of the function body should be alone on a line beginning in column 1. The matching closing brace should also be in column 1, and be followed by a comment giving the function name.

(6) Function scope variable declarations shall precede any code statements. There should be one local data declaration per line. The data types, variable names, and comments for each declaration should begin in the same column.

(7) A function should do one and only one thing, and it should be kept short.

(8) A function should not exceed two pages, excluding the function prologue. If a function exceeds this suggested limit, it may be performing too many "things" and the programmer should consider decomposing it into multiple functions. On the other hand, each function call has a run-time time and space overhead, and the impact of this overhead should also be considered.

(9) A function shall have a single entry point (no entry via *goto*) and it should have a single exit point. Multiple exit points are often convenient in "low-level" code, or in error conditions.

(10) A function should not have an excessive number of calling parameters. If a function has too many calling parameters, it may indicate that the function is too complex or performing too many actions. An alternative is to group the parameters into a struct; this also saves stack space at each function call if the struct is passed by reference.

(11) The number of logical nesting levels within a function should not exceed four. If a function has more, the programmer should consider decomposing it into multiple functions.

3.8 DATA TYPE DECLARATIONS

(1) All variables shall be explicitly typed.

(2) All types shall be declared with *typedef*, except that pointer types may be built from the base type by adding an asterisk.

(3) Structure tags shall not be used, except where required for forward references.

One item to consider is whether to use a structure of arrays or an array of structures. Quite often an array of structures will lead to a cleaner and more concise code. This is due to the fact that a structure provides cohesion among related data entities. An array of structures is a container of homogeneous data objects that can be operated upon in the same manner.

```
typedef struct /* This is a structure of arrays */
{
   double Variable1[2];
   float Variable2[2];
   short Variable3[2];
} StructArray;

typedef struct /* This is a structure */
{
   double Variable1;
   float Variable2;
   short Variable3;
} ArrayStruct;

ArrayStruct ArrayOfStruct[N]; /* this is an array of structures */
```

3.8.1 Bitfields

Bitfields provide an excellent abstraction for data items that are smaller than a word. They should be used whenever a structure contains such data items. Note that the order of declaration of bitfields is target-dependent; see Section 4. Some compilers have bugs in the implementation of bitfields.

An alternative is mask and shift macros; these can be defined in a target-independent way.

3.9 STATEMENTS

3.9.1 if Statements

(1) An if statement should have the following format:

```
if ( x == REQUEST)
{
   statement;
}
else
{
   statement;
}
```

The else clause is optional. The braces may be omitted for one-line statements.

(2) An *if-else* chain, in the form:

```
if (x == COMMAND)
{
   statement;
}
else if (x == REQUEST)
{
   statement;
}
else if (x == STATUS)
{
   statement;
}
```

should only be used when the conditions are all the same basic type (such as testing the same variable against different values), and the conditions involved are mutually exclusive. If the conditions are qualitatively different, the additional *if* statements should start on new lines, indented, as in:

```
if (x == REQUEST)
{
   statement;
}
else
{
   if (y == COMMAND)
   {
      statement;
   }
   else
   {
      statement;
   }
}
```

Because a switch statement is easier to understand, it should be used instead of *if-else* chains whenever possible.

(3) When testing a pointer for NULL, "`if (pointer == NULL)`" should be used, rather than "`if (!pointer)`". This makes the intent clearer, and guards against the (remote) possibility that some target may use a non-zero value for NULL.

3.9.2 switch Statement

(1) A switch statement should have the following format:

```
switch ( expression) {
case constant_expression_1 :
   statements;
   break;

case constant_expression_2 : /* falls thru */
case constant_expression_3 :
   statements;
   break;

default:
   statements;
   break;

} /* End of switch ( expression) */
```

Note that the opening brace is *not* on a separate line; this is an exception to the general rule.

(2) If one case falls through to the next case, make sure that it is fully commented.

(3) The *default* case shall be present. A waiver may be granted for a situation where it is immediately obvious that the default case will never be taken, but the danger to maintenance must be considered when granting this waiver.

3.10 SIDE EFFECTS WITHIN EXPRESSIONS

Expressions whose primary purpose is to assign a value to a variable may also modify one or more other variables. Similarly, expressions whose primary purpose is to compute a boolean value (in an *if* or *while* statement) can also modify variables. These secondary operations are called *side effects*.

(1) Side effects within expressions should not be used.

(2) In the condition portion of an *if*, *for*, *while*, etc., side effects that extend beyond the guarded statement block should be minimized. That is, in a statement like:

```
if ((c = getchar()) != EOF)
{
   guarded_statements
}
other_statements
```

It is natural to think of the variable "c" being bound to a value only within the *guarded_statements*. It should not be used in the *other_statements*.

3.11 MACRO STATEMENTS

Macros should be used only when necessary. Overuse of macros can make code harder to read and maintain because the code no longer reads or behaves like standard C.

(1) *#include* statements shall include header files only. Body files may not be included in other body files.

(2) Conditional compilation should be used only when absolutely necessary. One situation where conditional compilation is allowed is multi-platform development environments that require different code for each target environment. Even then, it may be more appropriate to put each variant in a separate body file.

(3) Multi-statement *#define* macros should be avoided.

(4) *#define* macros should be used to define constants that are shared among many files. However, consider declaring a const object instead; this allows type checking and symbol exportation to the debugger, at the cost of some data memory.

(5) Parentheses should be used when defining macro expressions. Consider the following macro definition:

```
#define SQUARE(x) x*x
```

`SQUARE(5)` properly expands to `5*5`. However `SQUARE(z+1)` expands to `z+1*z+1` which is NOT correct. The macro `SQUARE` should be defined as

```
#define SQUARE(x) ((x)*(x))
```

(6) *#define* shall not be used to redefine the C syntax or reserved words. For example this rules out the use of any of the following:

```
#define begin  {
#define end    }
#define OR     ||
#define AND    &&
```

(7) *#define* shall not be used to define new types; use *typedef* instead.

(8) Fields of *#defines* should line up. For example,

```
#define  THIS       1
#define  WHATEVER   2
```

(9) *#include* statements shall not use absolute path names.

Including directory information impacts code portability. However, paths in *#include* are sometimes necessary. Examples are for system headers such as ‘sys/time.h’, and for third-party libraries that happen to use the same file names. In these cases, or others with sufficient justification, a waiver from this requirement can be obtained.

4 PORTABILITY

It is not always possible to write totally portable code; some targets will require different code to implement a particular function. To increase the ease of porting code to many targets, it is best to isolate the non-portable code in a few places, behind an interface that is portable.

In addition, the techniques listed here solve some portability issues. Always using these techniques to solve these particular issues will make it easier for readers to understand what the code is doing.

(1) All code in the branch library should be portable among the processors given in Table 2. The branch library shall document which processors each unit has been tested on.

This portability requirement allows projects more freedom in choosing processors, and allows running flight code in a simulator that uses a different processor. Unit tests are also often run on a different processor during development.

(2) When conditional compilation on the processor type is needed, the following structure shall be used:

```
#if defined (__ut69r__)
/* definitions for ut69r */

#elif defined (__ix86__)
/* definitions for x86 */

#else
#error undefined processor
#endif
```

The identifier for the *defined* macro is given in Table 2.

TABLE 2. Recommended processors

Processor	identifier
Goddard Mongoose	
Intel x86	`_ix86_`
Motorola PowerPC	
United Technologies ut69r	`_ut69r_`

Operating systems are another source of portability problems; this standard does not address this issue yet.

4.1 COMMON TYPES

(1) The following types shall be used when the size of data is important, such as when the data is shared between different computers:

```
typedef system_specific int8;
typedef system_specific int16;
```

```
typedef system_specific int32;
typedef system_specific uint8;
typedef system_specific uint16;
typedef system_specific uint32;
```

Data whose size is not important may use the standard C types *int*, *unsigned*, etc.

(2) The common types shall be defined in the header file "common_types.h", using appropriate processor-dependent conditional compilation.

(3) The type *char* shall only be used for data that is a string. *char* can be signed or unsigned, depending on the compiler. When used for strings, this difference is irrelevant. If a *char*-sized number is needed, use the appropriate common type.

4.2 BYTE ORDERING

There are two kinds of *endianness*; bit and byte. *Bit endianness* determines how data is stored in registers; *byte endianness* determines how data is stored in memory. Big-byte-endian means data is stored in memory with the highest-order byte at the lowest address. Little-byte-endian has the least significant byte at the lowest address. Big-bit-endian means data is stored in a register with the most significant bit in the lowest numbered bit. Little-bit-endian has the least significant bit in the lowest numbered bit.

The combination of bit and byte endianness affects the way C bitfields in structs are laid out in memory. The ANSI C standard says this is implementation dependent; the information given here is valid for the Borland C and C++ compilers for Intel x86 processors, and for the Gnu C compiler for Intel x86, Motorola PowerPC, and ut69r.

For struct fields that are not bit fields, the fields are laid out in memory with the first field at the lowest address, and following fields at subsequent addresses. However, there may be padding (unused space) to meet alignment requirements (*double* fields are often aligned on 8 byte boundaries).

For bit fields, the fields are laid out in *int* sized chunks; the first chunk at the lowest address, following chunks at subsequent addresses. Within a chunk, the combination of bit and byte endianness determines the layout order.

(1) For each processor, the "common_types.h" header file shall define one of the preprocessor constants `__STRUCT_HIGH_BIT_FIRST__` or `__STRUCT_LOW_BIT_FIRST__` to reflect the combination of bit and byte endianness for that processor.

(2) The following format shall be used to define bitfields in a portable way. Note that macros implementing mask/shift operations provide another alternative for portable bitfield access.

```
#if defined (__STRUCT_HIGH_BIT_FIRST__)

   typedef struct
   {
      /* word 0 */
      unsigned Word_0_High_Bits : 2;
      unsigned Word_0_Middle_Bits : 9;
      unsigned Word_0_Low_Bits : 5;
      /* word 1 */
      unsigned Word_1_High_Bits : 2;
      unsigned Word_1_Middle_Bits : 9;
      unsigned Word_1_Low_Bits : 5;
   } Example_Bitfield_type;

#elif defined (__STRUCT_LOW_BIT_FIRST__)

   typedef struct
   {
      /* word 0 */
      unsigned Word_0_Low_Bits : 5;
      unsigned Word_0_Middle_Bits : 9;
      unsigned Word_0_High_Bits : 2;
      /* word 1 */
      unsigned Word_1_Low_Bits : 5;
      unsigned Word_1_Middle_Bits : 9;
      unsigned Word_1_High_Bits : 2;
   } Example_Bitfield_type;

#else
#error struct bit order not defined
#endif
```

Note that the specific structure fields shown here are just an example; it is the *#if #elif #else #endif* structure that is required.

Note that the ut69r compiler limits bit fields to 16 bits, since that is the size of *int*.

4.3 STRUCTURE PACKING

(1) Compiler options to force structures to be packed shall only be used when necessary to make data that is shared between different compilers or targets compatible.